Typing a Core Binary Field Arithmetic in a Light Logic

Emanuele Cesena[1]*, Marco Pedicini[2], and Luca Roversi[3]

[1] Politecnico di Torino, Dip. di Automatica e Informatica, Torino, Italy
emanuele.cesena@polito.it
[2] Istituto per le Applicazioni del Calcolo "Mauro Picone", CNR, Roma, Italy and
LIPN – UMR CNRS 7030, Institut Galilée - Université Paris-Nord
marco@iac.cnr.it
[3] Università degli Studi di Torino, Dip. di Informatica, Torino, Italy
roversi@di.unito.it

Abstract. We design a library for binary field arithmetic and we supply a core API which is completely developed in DLAL, extended with a fix point formula. Since DLAL is a restriction of linear logic where only functional programs with polynomial evaluation cost can be typed, we obtain the core of a functional programming setting for binary field arithmetic with *built-in* polynomial complexity.

1 Introduction

Embedded systems (smart cards, mobile phones, sensors) are very heavy on resources. Low memory and computational power force programmers to choose specific algorithms and fine tune them in order to carefully manage the space and time complexity. There is an applicative domain where these constraints on resources cause serious difficulties: the implementation of cryptographic primitives, that is the foundation for strong security mechanisms and protocols.

We have started reasoning about a controlled programming setting, that should enable the certification of resource usage (memory and computation time), in a functional programming language. We are aware of different approaches to solve analogous problems, for instance the Computer Aided Cryptography Engineering (CACE) European project[4] whose mission is "to enable verifiable secure cryptographic software engineering to non-experts by developing a toolbox which automatically produces high-performance solutions from natural specifications".

What if difficulties on time/space complexity were *automagically* overcome by imposing an appropriate type discipline to the programming language?

This paper describes preliminary results on how to devise a programming language that grants a natural programming style in the implementation of specific number theoretic algorithms, in combination with a type discipline which ensures complexity bounds. More precisely, we investigate how to achieve implementation of number theoretic algorithms with certified running time bounds by exploiting logical tools under the prescriptions of Implicit Computational Complexity (ICC) [1]. We recall that ICC

* Partially supported by the European Project TClouds (http://www.tclouds-project.eu).
[4] http://www.cace-project.eu

mainly aims at searching strong mathematical roots for computational complexity theory. The logical approach to ICC extracts functional language primitives from logical systems under the Curry-Howard analogy. The logical system for ICC we focus on is DLAL [2]. It derives from linear logic. Its formulas can be types of λ-terms. A λ-term M typable in DLAL reduces to its normal form in a time which is a polynomial in the dimension of M.

We propose to put this theory into practice by developing and implementing a *core library* of combinators, namely λ-terms, typeable in DLAL. The library currently implements a subset of functionalities which are needed for binary field arithmetic (cf., e.g., [3, Section 11.2]). The practical relevance of completing such a library is to import functional programming technology with a known predetermined complexity into the area of applied cryptography.

Contributions. Defining a core library that correctly implements finite field arithmetic is a result in itself. The reason is that when programming non obvious combinators typeable in DLAL, the main obstacle lies in the application of the standard *divide-et-impera* paradigm: first *split* the problem into successively simpler ones until the solution becomes trivial, then *compose* the results. Composition is the harmful activity as soon as we face complexity issues. For example, using the output of a sub-problem, which results from an iteration, as the input of another iteration may yield a computational complexity blowup. This is why, in DLAL, naively manipulating lists by means of iterations, can rapidly "degrade" to situations where compositions which would be natural in standard λ-calculus simply get forbidden. It is for this reason that λ-terms in DLAL which implement the low level library with finite field operations are not the natural ones that we could write using λ-terms typeable in the System F [4].

To overcome the need of programming with non natural λ-terms, we follow [5], which promotes standard programming patterns to assure readability and soundness of functional programs. We build an experimental `API` on top of the core library, which exports standard programming patterns. The goal of supplying an `API` is to help non experts writing λ-terms which are not directly typeable in DLAL, but which, roughly speaking, can be checked to compile into λ-terms with a type in DLAL.

Related Works on Polynomial Time Languages. A programming language inspired by Haskell is described in [6]. The programs that can be developed in it belong to the class of polynomial time functions because the language inherits the principles of the λ-terms, or, equivalently, of the proof-nets of LAL [7]. However, we are not aware of any attempt to exploit it to program libraries with a real potential impact. The approach of [6] to the development of a real programming language for polynomial time computations is quite orthogonal to ours. We proceed bottom-up, showing that a reasonably interesting library can be developed inside DLAL. Then, we import standard programming patterns which were compatible with the typing discipline of DLAL. In [6], the language is given under the assumption that its primitives will really be used.

The same occurs in [8] and [9]. The former extends λ-calculus to give formulas of SLL [10]. The latter introduces POLA, a programming language which mixes object

oriented and recursion schemes for which an interpreter is also available[5]. The best developed project we are aware of, and which brings theoretical results related to the world of polynomial time bounded functions "down to" the practical level, is based on [11, 12]. The language exploits formulas of a smartly crafted version of multiplicative linear logic as types and is based on recursion schemes à la System T. We are still far from those levels of migration of theory to practice.

Our main distinguishing feature is to remain loyal to the theoretical properties of DLAL, while allowing programming with standard patterns of functional programming.

2 Typed Functional Assembly

λ-calculus. Given a set $\mathcal{V}$, which we range over by *any lowercase Latin letter*, the set Λ of λ-terms, which we range over by the *uppercase Latin letters* M, N, P, Q, R, this set contains terms generated as follows:

$$M ::= \mathcal{V} \mid \lambda x.M \mid (M)\, M. \tag{1}$$

The set of free variables in M is $\mathrm{fv}(M)$. The set Λ^{v} of *values of our computations*, which we range over by *the uppercase Latin letters* V, W, X, is defined as follows:

$$V ::= \mathcal{V} \mid \lambda x.V \mid (x)\, V. \tag{2}$$

We remark that Λ^{v} coincides the standard β *normal forms*.

$$\frac{}{x \Downarrow x}\ \mathrm{v} \qquad \frac{M \Downarrow V}{\lambda x.M \Downarrow \lambda x.V}\ \mathrm{f} \qquad \frac{M \Downarrow x \quad N \Downarrow V}{(M)\, N \Downarrow (x)\, V}\ @\mathrm{v} \qquad \frac{M \Downarrow \lambda x.V \quad V\{^{N}/_{x}\} \Downarrow W}{(M)\, N \Downarrow W}\ @\mathrm{l}$$

Fig. 1. Big steps rewriting relation $\Downarrow$ on Λ with results in Λ^{v}

Big Steps Rewriting Relation on λ-calculus. The relation $\Downarrow \subset \Lambda \times \Lambda^{\mathrm{v}}$ is inductively defined in Figure 1.

2.1 Type assignment

We introduce a type assignment TFA which gives formulas of Linear Logic as types to λ-terms. In fact, TFA is DLAL [2] whose set of formulas is quotiented by a specific recursive equation. We recall that adding a recursive equation among the formulas does not negatively affect polynomial time soundness of DLAL normalization which only depends on the structural constraints that the process of formula construction puts on the form of derivations [1].

[5] http://projects.wizardlike.ca/projects/pola

$$\frac{}{\emptyset \mid x:A \vdash x:A}\ \mathrm{a} \qquad \frac{\Delta \mid \Gamma \vdash M:A}{\Delta,\Delta' \mid \Gamma,\Gamma' \vdash M:A}\ \mathrm{w} \qquad \frac{\Delta, x:A, y:A \mid \Gamma \vdash M:B}{\Delta, z:A \mid \Gamma \vdash M\{^z/_x\,^z/_y\}:B}\ \mathrm{c}$$

$$\frac{\Delta \mid \Gamma, x:A \vdash M:B}{\Delta \mid \Gamma \vdash \lambda x.M:A \multimap B}\ \multimap\mathrm{I} \qquad \frac{\Delta \mid \Gamma \vdash M:A \multimap B \quad \Delta' \mid \Gamma' \vdash N:A}{\Delta,\Delta' \mid \Gamma,\Gamma' \vdash (M)\,N:B}\ \multimap\mathrm{E}$$

$$\frac{\Delta, x:A \mid \Gamma \vdash M:B}{\Delta \mid \Gamma \vdash \lambda x.M:!A \multimap B}\ \Rightarrow\mathrm{I} \qquad \frac{\Delta \mid \Gamma \vdash M:!A \multimap B \quad \emptyset \mid \Delta' \vdash N:A \quad |\Delta'| \leq 1}{\Delta,\Delta' \mid \Gamma \vdash (M)\,N:B}\ \Rightarrow\mathrm{E}$$

$$\frac{\emptyset \mid \Delta, \Gamma \vdash M:A}{\Delta \mid \S\Gamma \vdash M:\S A}\ \S\mathrm{I} \qquad \frac{\Delta \mid \Gamma \vdash N:\S A \quad \Delta' \mid x:\S A, \Gamma' \vdash M:B}{\Delta,\Delta' \mid \Gamma,\Gamma' \vdash M\{^N/_x\}:B}\ \S\mathrm{E}$$

$$\frac{\Delta \mid \Gamma \vdash M:A \quad \alpha \notin \mathrm{fv}(\Delta,\Gamma)}{\Delta \mid \Gamma \vdash M:\forall\alpha.A}\ \forall\mathrm{I} \qquad \frac{\Delta \mid \Gamma \vdash M:\forall\alpha.A}{\Delta \mid \Gamma \vdash M:A[^B/_\alpha]}\ \forall\mathrm{E}$$

Fig. 2. Type assignment system TFA

Types for TFA. Given a set $\mathcal{G}$ of *formula variables*, which we range over by *lowercase Greek letters*, the set $\mathcal{F}$ of *formulas*, that we range over by the *uppercase Latin letters* A, B, C, D, is defined as follows:

$$A ::= \alpha \mid A \multimap A \mid !A \multimap A \mid \forall\alpha.A \mid \S A$$

Note that modal formulas $!A$ can occur in negative positions only. We obtain the set of *types* $\mathcal{T}$ when we consider the quotient of $\mathcal{F}$ by the following fix-point equation:

$$\mathbb{S} \equiv \forall\alpha.\mathbb{S}[\alpha] \tag{3}$$

where $\mathbb{S}[\alpha] \equiv (\mathbb{B}_2 \multimap \alpha) \multimap ((\mathbb{B}_2 \otimes \mathbb{S}) \multimap \alpha) \multimap \alpha$ and $\mathbb{B}_2$ is defined in Figure 3. We say $\mathbb{S}$ is the type of *Sequences*. Thus, we actually use formulas which are equivalence classes of types in $\mathcal{T}$.

Note that once we use $\mathbb{S}$ as type of a λ-term M, we can equivalently use any of its "unfolded forms" as type of M as well. In Figure 3, we also introduce relevant types we use to develop our first level library. As a notation, $A[^B/_\alpha]$ is the clash free substitution of B for every free occurrence of α in A (here, clash-free means that occurrences of free variables of B are not bound in $A[^B/_\alpha]$).

Type assignment TFA. We give the type assignment system TFA in Figure 2. In this formal system, we have judgments of the form $\Delta \mid \Gamma \vdash M : A$ where *context* Δ is *exponential*, while context Γ is *linear*. Any context is a finite domain function $x_1 : A_1, \ldots, x_n : A_n$ with domain $\{x_1, \ldots, x_n\}$, and range $\{A_1, \ldots, A_n\}$ in the codomain of the set of types. Every pair $x:A$ of any kind of context is a *type assignment for a variable*.

Tuples as primitives. The definition of tuples in Figure 3 supports the introduction of the tuples as primitives, as follows. Extending λ-calculus with tuples means adding the following clauses to (1):

$$M ::= \ldots \mid \langle M, \ldots, M\rangle \mid \lambda\langle x, \ldots, x\rangle.M. \tag{4}$$

Type	Definitions
Finite types	$\mathbb{B}_n[\alpha] \equiv \overbrace{\alpha \multimap \cdots \multimap \alpha}^{n+1} \multimap \alpha$ $\mathbb{B}_n \equiv \forall\alpha.\mathbb{B}_n[\alpha]$
Tuples	$(A_1 \otimes \ldots \otimes A_n)[\alpha] \equiv A_1 \multimap \cdots \multimap A_n \multimap \alpha$ $(A_1 \otimes \ldots \otimes A_n) \equiv \forall\alpha.(A_1 \otimes \ldots \otimes A_n)[\alpha] \multimap \alpha$
Church numerals	$\mathbb{U}[\alpha] \equiv !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ $\mathbb{U} \equiv \forall\alpha.\mathbb{U}[\alpha]$
Lists	$\mathbb{L}(A)[\alpha] \equiv !(A \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ $\mathbb{L}(A) \equiv \forall\alpha.\mathbb{L}(A)[\alpha]$
Church words	$\mathbb{L}_2 \equiv \mathbb{L}(\mathbb{B}_2)$

Fig. 3. Relevant (defined) types

So, values in (2) also include:

$$V ::= \ldots \mid \langle V, \ldots, V \rangle, \tag{5}$$

and the set of rules in Figure 1 must contain:

$$\frac{M_1 \Downarrow V_1 \quad \ldots \quad M_n \Downarrow V_n}{\langle M_1, \ldots, M_n \rangle \Downarrow \langle V_1, \ldots, V_n \rangle}\ \mathrm{p}$$

$$\frac{M \Downarrow \lambda\langle x_1, \ldots, x_n \rangle.V \quad N \Downarrow \langle V_1, \ldots, V_n \rangle \quad V\{^{V_1}/_{x_1}, \ldots, ^{V_n}/_{x_n}\} \Downarrow W}{(M)\, N \Downarrow W}\ @\mathrm{p}$$

Finally, we add the following derivable rules to those ones in Figure 2:

$$\frac{\varDelta_1 \mid \varGamma_1 \vdash M_1 : A_1 \quad \ldots \quad \varDelta_n \mid \varGamma_n \vdash M_n : A_n}{\varDelta_1 \ldots \varDelta_n \mid \varGamma_1 \ldots \varGamma_n \vdash \langle M_1, \ldots, M_n \rangle : (A_1 \otimes \ldots \otimes A_n)}\ \otimes\mathrm{I}$$

$$\frac{\varDelta \mid \varGamma, x_1 : A_1 \ldots x_n : A_n \vdash M : B}{\varDelta \mid \varGamma \vdash \lambda\langle x_1, \ldots, x_n \rangle.M : (A_1 \otimes \ldots \otimes A_n) \multimap B}\ \multimap\mathrm{I}_\otimes$$

Saying that the here above rules are derivable means that we use tuple as abbreviations, as follows:

$$\langle M_1, \ldots, M_n \rangle \equiv \lambda x.(\ldots((x)\, M_1) \ldots)\, M_n \tag{6}$$

$$\lambda\langle x_1, \ldots, x_n \rangle.M \equiv \lambda p.(p)\, \lambda x_1. \ldots \lambda x_n.M \tag{7}$$

3 A Library for Binary Field Arithmetic

In this section, we present a library of lambda-terms for the arithmetic in binary fields written in DLAL. The library is organized in functional layers, as shown in Figure 4.

The lowest layer contains basic definitions and it is interpreter-specific. We have currently implemented the library with LCI[6], an interpreter for pure λ-calculus. We thus needed to define basic types, such as Church words, or DLAL-specific combinators. The *core library* layer contains all the combinators to work on basic types. We put particular care in the definition of common functional-programming patterns in DLAL, and to reuse them, whenever possible, while defining other combinators. Finally, in the *binary field arithmetic* layer we group all the combinators related to operations over binary polynomials, like addition, multiplication and modular reduction.

In future work, we plan to extend the library by implementing other layers, such as arithmetic of elliptic curves or other cryptographic primitives, on top of the binary field arithmetic layer.

Cryptographic primitives*: elliptic curves cryptography, . . .*
Binary field arithmetic: addition, (modular reduction), square, multiplication, *inversion*.
Core library: operations on bits (xor, and), operations on sequences (head-tail splitting), operations on words (reverse, drop, conversion to sequence, projections); meta-combinators: fold, map, mapthread, map with state.
Basic definitions and types: booleans, tuples, numerals, words, sequences, basic type management and duplication.

Fig. 4. Library for binary field arithmetic

In the following subsections we present type and behaviour of the relevant combinators, while the full definition as λ-term is in Appendix A.

3.1 Basic Definitions and Types

In Figure 5, we give names to those formulas which are *types* we actually use in the library and we identify the λ-terms that we define as *canonical values* of the corresponding type. In every Sequence $[\mathsf{b}_{n-1} \ldots \mathsf{b}_0]$ and Church word $\{\mathsf{b}_{n-1} \ldots \mathsf{b}_0\}$ the *least significant bit* (l.s.b.) is b_0, while the *most significant bit* (m.s.b.) is b_{n-1}.
In DLAL, we can derive the rule *paragraph lift*:

$$\frac{\emptyset \mid \emptyset \vdash M : A \multimap B}{\emptyset \mid \emptyset \vdash \S[M] : \S A \multimap \S B}\ \S\mathrm{L}$$

where $\S[M] \equiv \lambda x.(M)\, x$ is the *paragraph lift of* M. As obvious generalization, n consecutive applications of the §L rule define a lifted term $\S^n[M] \equiv \lambda x.(\ldots \lambda x.(M)\, x \ldots)\, x$, that contains n nested $\S[\cdot]$. Its type is $\S^n A \multimap \S^n B$. Borrowing terminology from proof

[6] http://lci.sourceforge.net

	(typed) Values
Booleans	$\mathtt{1} \equiv \lambda xyz.x : \mathbb{B}_2$ $\mathtt{0} \equiv \lambda xyz.y : \mathbb{B}_2$ $\perp \equiv \lambda xyz.z : \mathbb{B}_2$
Tuples	$\langle M_1, \ldots, M_n\rangle \equiv \lambda p.(\ldots((p)\, M_1)\ldots)\, M_n : (A_1 \otimes \ldots \otimes A_n)$
Church numerals	$\mathtt{u}\varepsilon \equiv \lambda f x.x : \mathbb{U}$ $\overline{n} \equiv \lambda f x.\underbrace{(f)\ \ldots (f)}_{n}\ x : \mathbb{U}$
Church words	$\{\varepsilon\} \equiv \lambda f x.x : \mathbb{L}_2$ $\{\mathtt{b}_{n-1} \ldots \mathtt{b}_0\} \equiv \lambda f x.((f)\, \mathtt{b}_{n-1})\ \ldots ((f)\, \mathtt{b}_0)\, x : \mathbb{L}_2$
Sequences	$[\varepsilon] \equiv \lambda tc.(t) \perp : \mathbb{S}$ $[\mathtt{b}_{n-1} \ldots \mathtt{b}_0] \equiv \lambda tc.(c)\, \langle \mathtt{b}_{n-1}, [\mathtt{b}_{n-2} \ldots \mathtt{b}_0]\rangle : \mathbb{S}$

Fig. 5. Canonical values of data-types

nets, the application of n paragraph lift of M *embeds* it in n paragraph boxes, leaving the behaviour of M unchanged:

$$(\S^n[M])\, N \Downarrow (M)\, N.$$

The combinator $\mathtt{bCast}^m : \mathbb{B}_2 \multimap \S^{m+1}\mathbb{B}_2$ embeds a boolean into $m + 1$ paragraph boxes, without altering the boolean:

$$(\mathtt{bCast}^m)\, \mathtt{b} \Downarrow \mathtt{b}.$$

The combinator $\mathtt{b}\nabla_t : \mathbb{B}_2 \multimap (\overbrace{\mathbb{B}_2 \otimes \cdots \otimes \mathbb{B}_2}^{t})$, for every $t \geq 2$, produces t copies of a boolean:

$$(\mathtt{b}\nabla_t)\, \mathtt{b} \Downarrow \langle \overbrace{\mathtt{b}, \ldots, \mathtt{b}}^{t} \rangle.$$

The combinator $\mathtt{tCast}^m : (\mathbb{B}_2 \otimes \mathbb{B}_2) \multimap \S^{m+1}(\mathbb{B}_2 \otimes \mathbb{B}_2)$, for every $m \geq 0$, embeds a pair of bits into $m + 1$ paragraph boxes, without altering the structure of the pair:

$$(\mathtt{tCast}^m)\, \langle \mathtt{b}_0, \mathtt{b}_1\rangle \Downarrow \langle \mathtt{b}_0, \mathtt{b}_1\rangle.$$

The combinator $\mathtt{wSuc} : \mathbb{B}_2 \multimap \mathbb{L}_2 \multimap \mathbb{L}_2$ implements the *successor* on Church words:

$$((\mathtt{wSuc})\, \mathtt{b})\, \{\mathtt{b}_{n-1} \ldots \mathtt{b}_0\} \Downarrow \{\mathtt{b}\ \mathtt{b}_{n-1} \ldots \mathtt{b}_0\}.$$

The combinator $\mathtt{wCast}^m : \mathbb{L}_2 \multimap \S^{m+1}\mathbb{L}_2$, for every $m \geq 0$, embeds a word into $m+1$ paragraph boxes, without altering the structure of the word:

$$(\mathtt{wCast}^m)\,\{\mathtt{b}_{n-1}\ldots\mathtt{b}_0\} \Downarrow \{\mathtt{b}_{n-1}\ldots\mathtt{b}_0\}.$$

The combinator $\mathtt{w}\nabla^m_t : \mathbb{L}_2 \multimap \S^{m+1}(\overbrace{\mathbb{L}_2\otimes\cdots\otimes\mathbb{L}_2}^{t})$, for every $t \geq 2$, $m \geq 0$, produces t copies of a word deepening the result into $m+1$ paragraph boxes:

$$(\mathtt{w}\nabla^m_t)\,\{\mathtt{b}_{n-1}\ldots\mathtt{b}_0\} \Downarrow \langle\overbrace{\{\mathtt{b}_{n-1}\ldots\mathtt{b}_0\},\ldots,\{\mathtt{b}_{n-1}\ldots\mathtt{b}_0\}}^{t}\rangle.$$

3.2 Core Library

Operations on Bits.
The combinator $\mathtt{Xor} : \mathbb{B}_2 \multimap \mathbb{B}_2 \multimap \mathbb{B}_2$ extends the *exclusive or* as follows:

$$\begin{array}{lll}
((\mathtt{Xor})\mathtt{0})\mathtt{0} \Downarrow \mathtt{0} & ((\mathtt{Xor})\mathtt{1})\mathtt{1} \Downarrow \mathtt{0} & \\
((\mathtt{Xor})\mathtt{0})\mathtt{1} \Downarrow \mathtt{1} & ((\mathtt{Xor})\mathtt{1})\mathtt{0} \Downarrow \mathtt{1} & \\
((\mathtt{Xor})\bot)\mathtt{b} \Downarrow \mathtt{b} & ((\mathtt{Xor})\mathtt{b})\bot \Downarrow \mathtt{b} & (\text{where } \mathtt{b} : \mathbb{B}_2).
\end{array}$$

Whenever one argument is $\bot$ then it gives back the other argument. This is an application oriented choice. Later we shall see why.
The combinator $\mathtt{And} : \mathbb{B}_2 \multimap \mathbb{B}_2 \multimap \mathbb{B}_2$ extends the *and* as follows:

$$\begin{array}{lll}
((\mathtt{And})\mathtt{0})\mathtt{0} \Downarrow \mathtt{0} & ((\mathtt{And})\mathtt{1})\mathtt{1} \Downarrow \mathtt{1} & \\
((\mathtt{And})\mathtt{0})\mathtt{1} \Downarrow \mathtt{0} & ((\mathtt{And})\mathtt{1})\mathtt{0} \Downarrow \mathtt{0} & \\
((\mathtt{And})\bot)\mathtt{b} \Downarrow \bot & ((\mathtt{And})\mathtt{b})\bot \Downarrow \bot & (\text{where } \mathtt{b} : \mathbb{B}_2).
\end{array}$$

Whenever one argument is $\bot$ then the result is $\bot$. Again, this is an application oriented choice.

Operations on Sequences.
The combinator $\mathtt{sSpl} : \mathbb{S} \multimap (\mathbb{B}_2 \otimes \mathbb{S})$ *splits* the sequence it takes as input in a pair with the m.s.b. and the corresponding tail:

$$(\mathtt{sSpl})\,[\mathtt{b}_{n-1}\ldots\mathtt{b}_0] \Downarrow \langle\mathtt{b}_{n-1}, [\mathtt{b}_{n-2}\ldots\mathtt{b}_0]\rangle.$$

Operations on Church Words.
The combinator $\mathtt{wRev} : \mathbb{L}_2 \multimap \mathbb{L}_2$ *reverses the bits* of a word:

$$(\mathtt{wRev})\,\{\mathtt{b}_{n-1}\ldots\mathtt{b}_0\} \Downarrow \{\mathtt{b}_0\ldots\mathtt{b}_{n-1}\}.$$

The combinator $\mathtt{wDrop}\bot : \mathbb{L}_2 \multimap \mathbb{L}_2$ *drops* all the (initial) occurrences[7] of $\bot$ in a word:

$$(\mathtt{wDrop}\bot)\,\{\bot\ldots\bot\ \mathtt{b}_{n-1}\ldots\mathtt{b}_0\} \Downarrow \{\mathtt{b}_{n-1}\ldots\mathtt{b}_0\}.$$

[7] The current definition actually drops all the occurrences of $\bot$ in a Church word, however we shall only apply $\mathtt{wDrop}\bot$ to words that contain $\bot$ in the most significant bits.

The combinator $\texttt{w2s} : \mathbb{L}_2 \multimap \S\mathbb{S}$ *casts* a word into a sequence:

$$(\texttt{w2s})\,\{\texttt{b}_{n-1}\ldots\texttt{b}_0\} \Downarrow [\texttt{b}_{n-1}\ldots\texttt{b}_0].$$

The combinator $\texttt{wProj} : \mathbb{L}(\mathbb{B}_2^2) \multimap \mathbb{L}_2$ *projects* the first component of a list of pairs:

$$(\texttt{wProj})\,\lambda f x.((f)\,\langle\texttt{a}_{n-1},\texttt{b}_{n-1}\rangle)\ \ldots((f)\,\langle\texttt{a}_0,\texttt{b}_0\rangle)\,x \Downarrow \{\texttt{a}_{n-1}\ldots\texttt{a}_0\}\ .$$

Similarly, $\texttt{wProj2} : \mathbb{L}(\mathbb{B}_2^2) \multimap \mathbb{L}_2$ projects the second component. The argument of $\texttt{wProj}$ has not the form $\{\langle\texttt{a}_{n-1},\texttt{b}_{n-1}\rangle\ldots\langle\texttt{a}_0,\texttt{b}_0\rangle\}$ because its elements are not booleans. We shall adopt the same convention also for the forthcoming meta-combinators.

Meta-combinators on Lists. Meta-combinators are λ-terms with one or two "holes" that allow to use standard higher-order programming patterns to extend the $\texttt{API}$. Holes must be filled with type constrained λ-terms. We discuss how to use meta-combinators in order to effectively implement arithmetic in Section 3.3, after their introduction here below.

The first meta-combinator we deal with is $\texttt{Map}[\cdot]$. Let $\texttt{F} : A \multimap B$ be a closed term. Then, $\texttt{Map}[\texttt{F}] : \mathbb{L}(A) \multimap \mathbb{L}(B)$ applies $\texttt{F}$ to every element of the list that $\texttt{Map}[\texttt{F}]$ takes as argument, and yields the final list, assuming $(\texttt{F})\,\texttt{b}_i \Downarrow \texttt{b}'_i$, for every $0 \le i \le n-1$:

$$(\texttt{Map}[\texttt{F}])\,\lambda f x.((f)\,\texttt{b}_{n-1})\ \ldots((f)\,\texttt{b}_0)\,x \Downarrow \lambda f x.((f)\,\texttt{b}'_{n-1})\ \ldots((f)\,\texttt{b}'_0)\,x$$

The second meta-combinator is $\texttt{Fold}[\cdot,\cdot]$. Let $\texttt{F} : A \multimap B \multimap B$ and $\texttt{S} : B$ be closed terms. Let also $\texttt{Cast}^0 : B \multimap \S B$. Then, $\texttt{Fold}[\texttt{F},\texttt{S}] : \mathbb{L}(A) \multimap \S B$, starting from the initial value $\texttt{S}$, iterates $\texttt{F}$ over the input list and builds up a value, assuming $((\texttt{F})\,\texttt{b}_i)\,\texttt{b}'_i \Downarrow \texttt{b}'_{i+1}$, for every $0 \le i \le n-1$, and setting $\texttt{b}'_0 = \texttt{S}$ and $\texttt{b}'_n = \texttt{b}'$:

$$(\texttt{Fold}[\texttt{F},\texttt{S}])\,\lambda f x.((f)\,\texttt{b}_{n-1})\ \ldots((f)\,\texttt{b}_0)\,x \Downarrow \texttt{b}'$$

The third meta-combinator is $\texttt{MapState}[\cdot]$. Let $\texttt{F} : (A \otimes S) \multimap (B \otimes S)$ be a closed term. Then, $\texttt{MapState}[\texttt{F}] : \mathbb{L}(A) \multimap S \multimap \mathbb{L}(B)$ applies $\texttt{F}$ to the elements of the input list, keeping track of a *state* of type S during the iteration. Specifically, if $(\texttt{F})\,\langle\texttt{b}_i,\texttt{s}_i\rangle \Downarrow \langle\texttt{b}'_i,\texttt{s}_{i+1}\rangle$, for every $0 \le i \le n-1$:

$$((\texttt{MapState}[\texttt{F}])\,\lambda f x.((f)\,\texttt{b}_{n-1})\ \ldots((f)\,\texttt{b}_0)\,x)\,\texttt{s}_0 \Downarrow \lambda f x.((f)\,\texttt{b}'_{n-1})\ \ldots((f)\,\texttt{b}'_0)\,x$$

Finally, the fourth meta-combinator is $\texttt{MapThread}[\cdot]$. Let $\texttt{F} : \mathbb{B}_2 \multimap \mathbb{B}_2 \multimap A$ be a closed term. Then, $\texttt{MapThread}[\texttt{F}] : \mathbb{L}_2 \multimap \mathbb{L}_2 \multimap \mathbb{L}(A)$ applies $\texttt{F}$ to the elements of the input list. Specifically, if $((\texttt{F})\,\texttt{a}_i)\,\texttt{b}_i \Downarrow \texttt{c}_i$, for every $0 \le i \le n-1$:

$$((\texttt{MapThread}[\texttt{F}])\,\{\texttt{a}_{n-1}\ldots\texttt{a}_0\})\,\{\texttt{b}_{n-1}\ldots\texttt{b}_0\} \Downarrow \lambda f x.((f)\,\texttt{c}_{n-1})\ \ldots((f)\,\texttt{c}_0)\,x$$

In particular, $\texttt{MapThread}[\lambda a.\lambda b.\langle a,b\rangle] : \mathbb{L}_2 \multimap \mathbb{L}_2 \multimap \mathbb{L}(\mathbb{B}_2^2)$ is such that:

$$\begin{aligned}((\texttt{MapThread}[\lambda a.\lambda b.\langle a,b\rangle])\,\{\texttt{a}_{n-1}\ldots\texttt{a}_0\})\,\{\texttt{b}_{n-1}\ldots\texttt{b}_0\} \Downarrow& \\ \lambda f x.((f)\,\langle\texttt{a}_{n-1},\texttt{b}_{n-1}\rangle)\ \ldots((f)\,\langle\texttt{a}_0,\texttt{b}_0\rangle)\,x&\end{aligned}$$

3.3 Binary Field Arithmetic

We start by recalling the essentials on binary field arithmetic. For wider details we address the reader to [3, Section 11.2]. Let $p(X) \in \mathbb{F}_2[X]$ be an irreducible polynomial of degree n over $\mathbb{F}_2$, and let $\beta \in \overline{\mathbb{F}}_2$ be a root of $p(X)$ in the algebraic closure of $\mathbb{F}_2$. Then, the finite field $\mathbb{F}_{2^n} \simeq \mathbb{F}_2[X]/(p(X)) \simeq \mathbb{F}_2(\beta)$.

The set of elements $\{1, \beta, \dots, \beta^{n-1}\}$ is a basis of $\mathbb{F}_{2^n}$ as a vector space over $\mathbb{F}_2$ and we can represent a generic element of $\mathbb{F}_{2^n}$ as a polynomial in β of degree lower than n:

$$\mathbb{F}_{2^n} \ni a = \sum_{i=0}^{n-1} a_i \beta^i = a_{n-1}\beta^{n-1} + \cdots + a_1\beta + a_0 \quad , \qquad a_i \in \mathbb{F}_2 \quad .$$

Moreover, the isomorphism $\mathbb{F}_{2^n} \simeq \mathbb{F}_2[X]/(p(X))$ allows us to implement the arithmetic of $\mathbb{F}_{2^n}$ relying on the arithmetic of $\mathbb{F}_2[X]$ and reduction modulo $p(X)$.

Since each element $a_i \in \mathbb{F}_2$ can be encoded as a bit, we can represent each element of $\mathbb{F}_{2^n}$ as a Church word of bits of type $\mathbb{L}_2$.

In what follows, we denote by $\boldsymbol{n}$ the Church numeral representing $n = \deg p(X)$, and by $\boldsymbol{p}$ the Church word: $\boldsymbol{p} \equiv \{p_n \dots p_0 \underbrace{\bot \dots \bot}_{n-1}\}$, where p_i are such that $p(X) = \sum p_i X^i$. Note that $\boldsymbol{p}$ has length $2n-1$. The $\bot$ in the least significative part are included for technical reasons, to simplify the discussion later.

Addition. Let $a, b \in \mathbb{F}_{2^n}$. The addition $a + b$ is computed component-wise, i.e., setting $a = \sum a_i\beta^i$ and $b = \sum b_i\beta^i$, then $a + b = \sum(a_i + b_i)\beta^i$. The sum $(a_i + b_i)$ is done in $\mathbb{F}_2$ and corresponds to the bitwise exclusive or. This led us to the following definition:
The combinator $\texttt{Add} : \mathbb{F}_{2^n} \multimap \mathbb{F}_{2^n} \multimap \mathbb{F}_{2^n}$ is:

$$\texttt{Add} \equiv \texttt{MapThread}[\texttt{Xor}] \tag{8}$$

Modular Reduction. Reduction modulo $p(X)$ is a fundamental building block to keep the size of the operands constrained. We implemented a naïf left-to-right method, assuming that: (1) both $p(X)$ and $n = \deg p(X)$ are fixed (thus axioms); (2) the length of the input is $2n$, i.e. we need exactly n repetitions of a basic iteration.
The combinator $\texttt{wMod}[\boldsymbol{n}, \boldsymbol{p}] : \mathbb{L}_2 \multimap \S\mathbb{F}_{2^n}$ is:

$$\begin{aligned}\texttt{wMod}[\boldsymbol{n}, \boldsymbol{p}] \equiv \lambda d.(\S[\texttt{wModEnd}])& \\ ((\boldsymbol{n})\, \lambda l.((\texttt{MapState}[\texttt{wModFun}])\, l)\, \langle \bot, \mathbb{0} \rangle)\, &(\texttt{wModBase}[\boldsymbol{p}])\, (\texttt{wCast}^0)\, d\end{aligned}$$

where:

$$\begin{aligned}
\texttt{wModEnd} &\equiv \lambda l.(\texttt{wDrop}\bot)\,(\texttt{wRev})\,(\texttt{wProj})\, l \\
\texttt{wModFun} &\equiv \lambda\langle e, s\rangle.(\lambda\langle d, p\rangle.(\lambda\langle s_0, s_1\rangle. \\
&\qquad (((((((s_0)\, \lambda dps.(\lambda\langle p', p''\rangle.\langle\langle((\texttt{Xor})\, d)\, p', s\rangle, \langle 1, p''\rangle\rangle)\, (\mathrm{b}\nabla_2)\, p)\,) \\
&\qquad\qquad \lambda dps.\langle\langle d, s\rangle, \langle \mathbb{0}, p\rangle\rangle) \\
&\qquad\qquad \lambda dps.\langle\langle \bot, s\rangle, \langle d, p\rangle\rangle)\, d)\, p)\, s_1 \\
&\qquad)\, s)\, e \\
\texttt{wModBase}[\boldsymbol{p}] &\equiv \lambda d.((\texttt{MapThread}[\lambda a.\lambda b.\langle a, b\rangle])\, (\texttt{wRev})\, d)\, (\texttt{wRev})\, \boldsymbol{p}
\end{aligned}$$

The basic iteration is implemented via $\mathtt{MapState}[\cdot]$, that operates on a list of bit pairs $\{\ldots\langle \mathtt{d}_i, \mathtt{p}_i\rangle\ldots\}$, where $\mathtt{d}_i$ are the bits of the input and $\mathtt{p}_i$ the bits of $\boldsymbol{p}$. The core of the algorithm is the combinator $\mathtt{wModFun} : (\mathbb{B}_2^2 \otimes \mathbb{B}_2^2) \multimap (\mathbb{B}_2^2 \otimes \mathbb{B}_2^2)$, that behaves as follows:

$$((\mathtt{wModFun})\ \underbrace{\langle \mathtt{d}_i, \mathtt{p}_i\rangle)}_{\text{elem. } e}\ \underbrace{\langle \mathtt{s}_0, \mathtt{p}_{i+1}\rangle}_{\text{status } s} \Downarrow \langle\underbrace{\langle \mathtt{d}_i{}', \mathtt{p}_{i+1}\rangle}_{e'}, \underbrace{\langle \mathtt{s}_0{}', \mathtt{p}_i\rangle}_{s'}\rangle\ ,$$

where $\mathtt{s}_0$ keeps the m.s.b. of $\{\ldots\mathtt{d}_i\ldots\}$ and it is used to decide wether to reduce or not at this iteration. Thus, $\mathtt{d}_i{}' = \mathtt{d}_i + \mathtt{p}_i$ if $\mathtt{s}_0 = \mathtt{1}$; $\mathtt{d}_i{}' = \mathtt{d}_i$ if $\mathtt{s}_0 = \mathtt{0}$; and $\mathtt{d}_i{}' = \perp$ when $\mathtt{s}_0 = \perp$ (that represents the initial state, when $\mathtt{s}_0$ still needs to be set).

Note that the second component of the status is used to shift $\boldsymbol{p}$ (right shift as the words have been reverted).

Square. Square in binary fields is a linear map (it is the absolute Frobenius automorphism). If $a \in \mathbb{F}_{2^n}$, $a = \sum a_i\beta^i$, then $a^2 = \sum a_i\beta^{2i}$. This operation is obtained by inserting zeros between the bits that represent a and leads to a polynomial of degree $2n-2$, that needs to be reduced modulo $p(X)$.

Therefore, we introduce two combinators: $\mathtt{wSqr} : \mathbb{L}_2 \multimap \mathbb{L}_2$ that performs the bit expansion, and $\mathtt{Sqr} : \mathbb{F}_{2^n} \multimap \S\mathbb{F}_{2^n}$ that is the actual square in $\mathbb{F}_{2^n}$. We have:

$$\mathtt{Sqr} \equiv \lambda a.(\mathtt{wMod}[\boldsymbol{n}, \boldsymbol{p}])\,(\mathtt{wSqr})\,a \tag{9}$$

and $\mathtt{wSqr} \equiv \lambda l f x.((l)\,\mathtt{wSqrStep}[f])\,x$, where $\mathtt{wSqrStep}[f] \equiv \lambda et.((f)\,\mathtt{0})\,((f)\,e)\,t$ has type $\mathbb{B}_2 \multimap \alpha \multimap \alpha$ if f is a non linear variable with type $\mathbb{B}_2 \multimap \alpha \multimap \alpha$.

Multiplication. Let $a, b \in \mathbb{F}_{2^n}$. The multiplication ab is computed as polynomial multiplication, i.e., with the usual definition, $ab = \sum_{j+k=i}(a_j + b_k)\beta^i$.

We currently implemented the naïve schoolbook method. A possible extension to the *comb method* is left as future straightforward work. On the contrary, it is not clear how to implement the Karatsuba algorithm, which reduces the multiplication of n-bit words to operations on $n/2$-bit words. The difficulty is to represent the splitting of a word in its half upper and lower parts.

Similarly as for the square, we have to distinguish between the polynomial multiplication $\mathtt{wMult} : \mathbb{L}_2 \multimap \mathbb{L}_2 \multimap \S\mathbb{L}_2$ and the field operation $\mathtt{Mult} : \mathbb{F}_{2^n} \multimap \mathbb{F}_{2^n} \multimap \S^2\mathbb{F}_{2^n}$, obtained by composing with the modular reduction. We have:

$$\begin{aligned}\mathtt{Mult} &\equiv \lambda ab.(\S[\mathtt{wMod}[\boldsymbol{n}, \boldsymbol{p}]])\,((\mathtt{wMult})\,a)\,b\\ \mathtt{wMult} &\equiv \lambda ab.(\S[\mathtt{wProj2}])\,((b)\,\lambda Ml.((\mathtt{wMultStep})\,\langle M, \perp\rangle)\,l)\,(\mathtt{wMultBase})\,(\mathtt{wCast}^0)\,a\end{aligned}$$

The internals of $\mathtt{wMult}$ are in Figure 6. It implements two nested iterations. The parameter b controls the external, and a the internal one.

The external iteration (controlled by b) works on words of bit pairs. The combinator $\mathtt{wMultStep} : \mathbb{B}_2^2 \multimap \mathbb{L}(\mathbb{B}_2^2) \multimap \mathbb{L}(\mathbb{B}_2^2)$ behaves as follows:

$$((\mathtt{wMultStep})\,\langle \mathtt{M}, \perp\rangle)\,\lambda f x.\ldots((f)\,\langle \mathtt{m}_i, \mathtt{r}_i\rangle)\ \ldots x \Downarrow \lambda f x.\ldots((f)\,\langle \mathtt{m}_{i-1}, \mathtt{r}_i'\rangle)\ \ldots x$$

$$
\begin{aligned}
\mathtt{wMultStep} &\equiv \lambda slfx.(\mathtt{wBMult}[f])\,((l)\,\mathtt{MSStep}[f,\mathtt{wFMult}])\,(\mathtt{MSBase}[x])\,(\mathtt{tCast}^0)\,s\\
\mathtt{wMultBase} &\equiv \lambda m.((\mathtt{MapThread}[\lambda a.\lambda b.\langle a,b\rangle])\,m)\,\{\varepsilon\}\\
\mathtt{MSStep}[f,\mathtt{wFMult}] &\equiv \lambda e.\lambda\langle w,s\rangle.(\lambda\langle e',s'\rangle.\langle((f)\,e')\,w,s'\rangle)\,((\mathtt{wFMult})\,e)\,s\\
\mathtt{MSBase}[x] &\equiv \lambda s.\langle x,s\rangle\\
\mathtt{wFMult} &\equiv \lambda\langle m,r\rangle.\lambda\langle M,\bar{m}\rangle.(\lambda\langle m',m''\rangle.(\lambda\langle M',M''\rangle.\\
&\qquad\quad \langle\langle\bar{m},((\mathtt{Xor})\,((\mathtt{And})\,m')\,M')\,r\rangle,\langle M'',m''\rangle\rangle\\
&\quad)\,(\mathrm{b}\nabla_2)\,m)\,(\mathrm{b}\nabla_2)\,M\\
\mathtt{wBMult}[f] &\equiv \lambda\langle w,s\rangle.(\lambda\langle M,\bar{m}\rangle.((f)\,\langle\bar{m},\mathbb{0}\rangle)\,w)\,s
\end{aligned}
$$

Fig. 6. Multiplication: definition of the combinators

where $\mathtt{M}$ is the current bit of the Multiplier b, and every $\mathtt{m}_i$ is a bit of the multiplicand a, and every $\mathtt{r}_i$ is a bit in the current result. The iteration is enabled by the combinator $\mathtt{wMultBase} : \mathbb{L}_2 \multimap \mathbb{L}(\mathbb{B}^2_2)$, that, on input a, creates $\lambda fx.((f)\,\langle\mathtt{m}_{n-1},\bot\rangle)\ \dots((f)\,\langle\mathtt{m}_0,\bot\rangle)\,x$, setting the initial bits of the result to $\bot$. The projection $\mathtt{wProj2}$ returns the result when the iteration stops.

The internal iteration is used to update the above list of bit pairs. The core of this iteration is the combinator $\mathtt{wFMult} : \mathbb{B}^2_2 \multimap \mathbb{B}^2_2 \multimap (\mathbb{B}^2_2 \otimes \mathbb{B}^2_2)$, that behaves as follows:

$$((\mathtt{wFMult})\ \underbrace{\langle\mathtt{m}_i,\mathtt{r}_i\rangle}_{\text{elem. } e})\ \underbrace{\langle\mathtt{M},\mathtt{m}_{i-1}\rangle}_{\text{status } s} \Downarrow \langle\underbrace{\langle\mathtt{m}_{i-1},\mathtt{M}\cdot\mathtt{m}_i+\mathtt{r}_i\rangle}_{e'},\underbrace{\langle\mathtt{M},\mathtt{m}_i\rangle}_{s'}\rangle\ .$$

For completeness, we list the type of the other combinators: $\mathtt{MSStep}[f,\mathtt{wFMult}] : \mathbb{B}^2_2 \multimap (\alpha\otimes\mathbb{B}^2_2)\multimap(\alpha\otimes\mathbb{B}^2_2)$, $\mathtt{MSBase}[x] : \mathbb{B}^2_2 \multimap (\alpha\otimes\mathbb{B}^2_2)$, $\mathtt{wBMult}[f] : (\alpha\otimes\mathbb{B}^2_2)\multimap\alpha$.

Inversion. It is under development. We are concentrating on the binary Euclidean algorithm, which is the "left-to-right" counterpart of the extended Euclidean algorithm (for a detailed analysis, we refer to Fong et al. [13]).

4 Developing (with) the Library

Beside the implementation of the library, we experimented the use of higher-order combinators to improve the readability of the code, as well as the programming experience. Inspired by [5], we have rewritten some combinators relying on standard programming pattern such as $\mathtt{Map}[\cdot]$ and $\mathtt{Fold}[\cdot,\cdot]$, "simulating" the behavior of a programmer that wants to add new functionality to the library. The idea is to let the programmer write a combinator in a more comfortable style, and then to *compile* the combinator to a value that admits a type in DLAL. In the following, we give some relevant examples of increasing difficulty.

We know that $\mathtt{w2s}$ is defined as $\mathtt{w2s} \equiv \lambda l.((l)\,\lambda estc.(c)\,\langle e,s\rangle)\,[\varepsilon]$. A programmer could anyway define it by using the standard programming pattern $\mathtt{Fold}[\cdot,\cdot]$ as follows:

$$\mathtt{w2sFromFold} \equiv \mathtt{Fold}[\lambda estc.(c)\,\langle e,s\rangle,[\varepsilon]]$$

The combinator `w2sFromFold` is a legal one because `w2sFromFold` compiles exactly to `w2s`. The compilation consists of in-line substituting the parameters of $\texttt{Fold}[\cdot,\cdot]$ and of applying the rewriting steps in Figure 1, whose key intermediate λ-terms are $\lambda l.((l)\,\lambda ez.(\lambda tc.(c)\,\langle e,z\rangle)\,e)\,[\varepsilon]$ and $\lambda l.((l)\,\lambda eztc.(c)\,\langle e,z\rangle)\,[\varepsilon]$.

As a second example, we consider the combinator $\texttt{wProj} \equiv \lambda l f x.((l)\,\lambda\langle a,b\rangle.(f)\,a)\,x$, we define the following combinator and we show that it is equivalent to the above one:

$$\texttt{wProjFromMap} \equiv \texttt{Map}[\lambda\langle a,b\rangle.a]$$

We recall that $\texttt{wProj} : \mathbb{L}(\mathbb{B}_2^2) \multimap \mathbb{L}_2$. While compiling the expression, we need the assumption that each element e of the input word is $\langle a',b'\rangle : \mathbb{B}_2^2$. The key step is the reduction from $\lambda l f x.((l)\,\lambda e.(f)\,(\lambda\langle a,b\rangle.a)\,e)\,x$ to $\lambda l f x.((l)\,\lambda\langle a',b'\rangle.(f)\,(\lambda\langle a,b\rangle.a)\,\langle a',b'\rangle)\,x$, by replacing $\langle a',b'\rangle$ for e in accordance with the assumption.

Finally, we show that the combinator $\texttt{Map[F]} \equiv \lambda l f x.((l)\,\lambda e.(f)\,(\texttt{F})\,e)\,x$ can be written using $\texttt{Fold}[\cdot,\cdot]$ (see also [5, Section 2]) as:

$$\texttt{MapFromFold[F]} \equiv \texttt{Fold}[\lambda epfx.((f)\,(\texttt{F})\,e)\,((p)\ f)\,x, \{\varepsilon\}]$$

Here, the compilation process shows that $(\texttt{Map[F]})\,l'$ and $(\texttt{MapFromFold[F]})\,l'$ are equivalent to the same value. We proceed by induction on the length of the Church word l'. First, we note that:

$$\texttt{MapFromFold[F]} \Downarrow \lambda l.((l)\,\lambda epfx.((f)\,(\texttt{F})\,e)\,((p)\ f)\,x)\,\{\varepsilon\}$$

The base case is easy to check: $(\texttt{Map[F]})\,\{\varepsilon\} \Downarrow \{\varepsilon\}$ and $(\texttt{MapFromFold[F]})\,\{\varepsilon\} \Downarrow \{\varepsilon\}$.

We now prove the inductive case. Let $l \equiv \lambda f x.((f)\,\texttt{b}_{n-1})\ \ldots ((f)\,\texttt{b}_0)\,x$ be a Church word of length n. Assume that $(\texttt{Map[F]})\,l \Downarrow V$ and $(\texttt{MapFromFold[F]})\,l \Downarrow V$. We want to show that `Map[F]` and `MapFromFold[F]` reduce to the same term for a Church word $l' \equiv \lambda f x.((f)\,\texttt{b})\,((l)\ f)\,x$ of length $n+1$. We report the key intermediate λ-terms:

$$\begin{aligned}
(\texttt{Map[F]})\,l' &\Downarrow \lambda f x.((l')\,\lambda e.(f)\,(\texttt{F})\,e)\,x \\
&\Downarrow \lambda f x.((f)\,(\texttt{F})\,\texttt{b})\,((l)\,\lambda e.(f)\,(\texttt{F})\,e)\,x && (10)\\
{\scriptstyle\text{ind. hyp.}} &\Downarrow \lambda f x.((f)\,(\texttt{F})\,\texttt{b})\,((V)\ f)\,x && (11)\\
(\texttt{MapFromFold[F]})\,l' &\Downarrow ((l')\,\lambda epfx.((f)\,(\texttt{F})\,e)\,((p)\ f)\,x)\,\{\varepsilon\} \\
&\Downarrow ((\lambda epfx.((f)\,(\texttt{F})\,e)\,((p)\ f)\,x)\,\texttt{b})\,((l)\,\lambda epfx.((f)\,(\texttt{F})\,e)\,((p)\ f)\,x)\,\{\varepsilon\} \\
{\scriptstyle\text{ind. hyp.}} &\Downarrow ((\lambda epfx.((f)\,(\texttt{F})\,e)\,((p)\ f)\,x)\,\texttt{b})\,V \\
&\Downarrow \lambda f x.((f)\,(\texttt{F})\,\texttt{b})\,((V)\ f)\,x
\end{aligned}$$

This example is particularly relevant because $\texttt{MapFromFold[F]} : \mathbb{L}(A) \multimap \S\mathbb{L}(B)$, and $\texttt{Map[F]} : \mathbb{L}(A) \multimap \mathbb{L}(B)$ compile to a common term despite their types differ. This is possible by applying two β-expansions from (10) to (11) which do not duplicate any structure.

5 Conclusion and Future Work

We have presented a core library for binary field arithmetic developed DLAL. The main motivation behind this work is to achieve a programming framework with *built-in* polynomial complexity and, from this perspective, this library is just a starting point, as

it lacks inversion and a complete realistic applicative example, such as elliptic curves cryptography. In the same line, the implementation of symmetric-key cryptographic algorithms (block/stream ciphers, hash functions, ...) looks attractive as well, thanks to the higher-order bitwise operations at the core of the current `API`.

Next, we shall investigate a full compilation process whose target will be machine code. Namely, we plan to go further beyond the first compilation phase of Section 4, where, in fact, we describe an in-line parameters unfolding of standard programming patterns like `Map`[·] and `Fold`[·, ·]. The compilation to machine code will target parallelization, generally implied by functional programming thanks to its reduced data dependency.

Interestingly, while programming the binary field arithmetic, we found that the main programming patterns we used can be assimilated to the MapReduce paradigm [14]. This means that not only DLAL can be used to certify polynomial-time complexity, but it is also suitable to be adapted to actual cloud platforms based on the MapReduce.

Finally, we do not exclude that more refined logics than DLAL can be used to realize a similar framework with even better built-in properties. Our choice of DLAL originated as a trade-off between flexibility in programming and constrains imposed by the typing system, but it is at the same time an experiment. Different logics can for instance measure the space complexity, or provide a more fine-grained time complexity.

References

1. Girard, J.Y.: Light linear logic. IC **143**(2) (1998) 175–204
2. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda calculus. IC **207**(1) (2009) 41–62
3. Avanzi, R.M., Cohen, H., Doche, C., Frey, G., Lange, T., Nguyen, K., Vercauteren, F.: Handbook of Elliptic and Hyperelliptic Curve Cryptography. CRC Press (2005)
4. Girard, J.Y., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge University Press (1989)
5. Hutton, G.: A tutorial on the universality and expressiveness of fold. JFP **9**(4) (1999) 355–372
6. Baillot, P., Gaboardi, M., Mogbil, V.: A polytime functional language from light linear logic. In Gordon, A.D., ed.: ESOP. Volume 6012 of LNCS., Springer (2010) 104–124
7. Asperti, A., Roversi, L.: Intuitionistic light affine logic. ACM ToCL **3**(1) (2002) 1–39
8. Baillot, P., Mogbil, V.: Soft lambda-calculus: A language for polynomial time computation. In Walukiewicz, I., ed.: FoSSaCS. Volume 2987 of LNCS., Springer (2004) 27–41
9. Burrell, J.M., Cockett, R., Redmond, F.B.: Pola: a language for PTIME programming. Accepted for presentation at LCC2009 (2009)
10. Lafont, Y.: Soft linear logic and polynomial time. TCS **318**(1-2) (2004) 163–180
11. Hofmann, M.: The strength of non-size increasing computation. In: POPL. (2002) 260–269
12. Hofmann, M.: Linear types and non-size-increasing polynomial time computation. IandC **183**(1) (2003) 57–85
13. Fong, K., Hankerson, D., Lopez, J., Menezes, A.: Field inversion and point halving revisited. IEEE Trans. Comput. **53**(8) (2004) 1047–1059
14. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51** (2008) 107–113

A Definition of Combinators

$\mathtt{bCast}^m$ is $\lambda b.(((b)\,1)\,\mathbb{0})\,\bot$.

$\mathtt{b}\nabla_t$ is $\lambda b.(((b)\,\langle\overbrace{1\ldots1}^{t}\rangle)\,\langle\overbrace{\mathbb{0}\ldots\mathbb{0}}^{t}\rangle)\,\langle\overbrace{\bot\ldots\bot}^{t}\rangle$, for every $t\geq 2$.

$\mathtt{tCast}^m$ is, for every $m\geq 0$:

$$\begin{aligned}\mathtt{tCast}^0 \equiv \lambda\langle a,b\rangle.((((a)\,&\lambda x.(((x)\,\langle 1,1\rangle)\,\langle 1,\mathbb{0}\rangle)\,\langle 1,\bot\rangle)\\ &\lambda x.(((x)\,\langle \mathbb{0},1\rangle)\,\langle \mathbb{0},\mathbb{0}\rangle)\,\langle \mathbb{0},\bot\rangle)\\ &\lambda x.(((x)\,\langle \bot,1\rangle)\,\langle \bot,\mathbb{0}\rangle)\,\langle \bot,\bot\rangle)\,b\\ \mathtt{tCast}^{m+1} \equiv \lambda p.&(\S[\mathtt{tCast}^m])\,(\mathtt{tCast}^0)\,p\ .\end{aligned}$$

$\mathtt{wSuc}$ is $\lambda bp.\lambda fx.((f)\,(\mathtt{bCast}^0)\,b)\,((p)\,f)\,x$.

$\mathtt{wCast}^m$ is, for every $m\geq 0$:

$$\begin{aligned}\mathtt{wCast}^0 &\equiv \lambda l.((((l)\,(\mathtt{wSuc})\,\mathbb{0})\,(\mathtt{wSuc})\,1)\,(\mathtt{wSuc})\,\bot)\,\{\varepsilon\}\\ \mathtt{wCast}^{m+1} &\equiv \lambda l.(\S[\mathtt{wCast}^m])\,(\mathtt{wCast}^0)\,l\ .\end{aligned}$$

$\mathtt{w}\nabla^m_t$, for every $t\geq 2$, and $m\geq 0$ is:

$$\begin{aligned}\mathtt{w}\nabla^0_t &\equiv \lambda l.(((l)\,(\mathtt{w\nabla Step})\,\mathbb{0})\,(\mathtt{w\nabla Step})\,1)\,\mathtt{w\nabla Base}\\ \mathtt{w}\nabla^{m+1}_t &\equiv \lambda l.(\S[\mathtt{w}\nabla^m_t])\,(\mathtt{w}\nabla^0_t)\,l\\ \mathtt{w\nabla Step} &\equiv \lambda \mathrm{b}.\lambda\langle x_1\ldots x_t\rangle.\langle\overbrace{((\mathtt{wSuc})\,\mathrm{b})\,x_1\ldots((\mathtt{wSuc})\,\mathrm{b})\,x_t}^{t}\rangle\\ \mathtt{w\nabla Base} &\equiv \langle\overbrace{\{\varepsilon\}\ldots\{\varepsilon\}}^{t}\rangle\ .\end{aligned}$$

$\mathtt{Xor}$ is: $\lambda bc.((((b)\,\lambda x.(((x)\,\mathbb{0})\,1)\,1)\,\lambda x.(((x)\,1)\,\mathbb{0})\,\mathbb{0})\,\lambda x.x)\,c$.
$\mathtt{And}$ is $\lambda bc.((((b)\,\lambda x.x)\,\lambda x.(((x)\,\mathbb{0})\,\mathbb{0})\,\bot)\,\bot)\,c$.
$\mathtt{sSpl}$ is $\lambda s.((s)\,\lambda t.\langle\bot,[\varepsilon]\rangle)\,\lambda x.x$.
$\mathtt{wRev}$ is $\lambda lfx.(((l)\,\mathtt{wRevStep}[f])\,\lambda x.x)\,x$ with:
  $\mathtt{wRevStep}[f]\equiv\lambda erx.(r)\,((f)\,e)\,x : \mathbb{B}_2\multimap(\alpha\multimap\alpha)\multimap\alpha\multimap\alpha$, when $f:\mathbb{B}_2\multimap\alpha\multimap\alpha$.
$\mathtt{wDrop}\bot$ is $\lambda lfx.((l)\,\lambda e.((((e)\,\lambda f.(f)\,1)\,\lambda f.(f)\,\mathbb{0})\,\lambda fz.z)\,f)\,x$.
$\mathtt{w2s}$ is $\lambda l.((l)\,\lambda estc.(c)\,\langle e,s\rangle)\,[\varepsilon]$.
$\mathtt{wProj}$ is $\lambda lfx.((l)\,\lambda\langle a,b\rangle.(f)\,a)\,x$.
$\mathtt{wProj2}$ is $\lambda lfx.((l)\,\lambda\langle a,b\rangle.(f)\,b)\,x$.
$\mathtt{Map[F]}$ is $\lambda lfx.((l)\,\lambda e.(f)\,(\mathtt{F})\,e)\,x$, with $\mathtt{F}:A\multimap B$ closed.
$\mathtt{Fold[F,S]}$ is $\lambda l.((l)\,\lambda ez.((\mathtt{F})\,e)\,z)\,(\mathtt{Cast}^0)\,\mathtt{S}$, with $\mathtt{F}:A\multimap B\multimap B$, and $\mathtt{S}:B$ closed.
$\mathtt{MapState[F]}$ is $\lambda lsfx.(\lambda\langle w,s'\rangle.w)\,((l)\,\mathtt{MSStep}[\mathtt{F},f])\,(\mathtt{MSBase}[x])\,(\mathtt{Cast}^0)\,s$, with
  $\mathtt{F}:(A\otimes S)\multimap(B\otimes S)$ closed, and:

$$\begin{aligned}\mathtt{MSStep}[\mathtt{F},f] &\equiv \lambda e.\lambda\langle w,s\rangle.(\lambda\langle e',s'\rangle.\langle((f)\,e')\,w,s'\rangle)\,(\mathtt{F})\,\langle e,s\rangle : (A\otimes S)\multimap(\alpha\otimes S)\multimap(\alpha\otimes S)\\ \mathtt{MSBase}[x] &\equiv \lambda s.\langle x,s\rangle : S\multimap(\alpha\otimes S)\ .\end{aligned}$$

$\mathtt{MapThread[F]}$ is $\lambda lmfx.(\lambda\langle w,s\rangle.w)\,((l)\,\mathtt{MTStep}[\mathtt{F},f])\,(\mathtt{MTBase}[x])\,(\mathtt{w2s})\,(\mathtt{wRev})\,m$, with
  $\mathtt{F}:A\multimap B\multimap C$ closed, $(\mathtt{w2s})\,(\mathtt{wRev})\,m:\S\mathbb{S}$, whenever $m:\mathbb{L}_2$, and:

$$\begin{aligned}\mathtt{MTStep}[\mathtt{F},f] &\equiv \lambda a.\lambda\langle w,s\rangle.(\lambda\langle b,s'\rangle.\langle((f)\,((\mathtt{F})\,a)\,b)\,w,s'\rangle)\,(\mathtt{sSpl})\,s : \mathbb{B}_2\multimap(\alpha\otimes\mathbb{S})\multimap(\alpha\otimes\mathbb{S})\\ \mathtt{MTBase}[x] &\equiv \lambda x.\langle x,m\rangle : \alpha\multimap\alpha\otimes\mathbb{S}\ .\end{aligned}$$